Apache Airflow: Controlando fluxos de trabalhos através de DAG's

Eduardo Luiz, Marcelo Luiz

Resumo—O alto volume, as diferentes fontes e a necessidade de transformação de dados, constitui um cenário complexo para geração de informação útil para os usuários e organizações. Criar, manter e monitorar fluxos de trabalhos, garantindo a correta execução das tarefas de processamento de informações, é um desafio para as empresas e times de desenvolvimento como um todo, principalmente nas arquiteturas de sistemas atuais que são distribuídos com ou sem a utilização de *Microserviços*. Nesse contexto, o *Apache Airflow* oferece uma infraestrutura pronta para criação e gerenciamento de fluxos de trabalho complexos. Este arigo tem como objetivo apresentar de forma rápida como funciona a plataforma através da implementação de modelos baseados em grafos do tipo DAG.

Palavras Chaves—Apache, Airflow, fluxo, grafo, dag, tarefas.

Abstract—The high volume, the different sources and the need for data transformation constitute a complex scenario for generating useful information for users and organizations. Creating, maintaining and monitoring workflows, ensuring correct execution of information processing tasks, is a challenge for enterprises and development teams as a whole, especially in current system architectures that are distributed with or without the use of textit Microservices. In this context, textit Apache Airflow provides an infrastructure ready for creating and managing complex workflows. This arigo aims to present quickly how the platform works through the implementation of models based on DAG type graphs.

Index Terms—Apache, Airflow, fluxo, grafo, dag, tarefas

1 Introdução

Fluxos de trabalho são mais comuns do que se imagina em aplicações corporativas. Muitas atividades das organizações modernas se baseiam em um conjunto de processamentos de informações que podem estar restritos a um único sistema, ou na maioria dos casos, estar distribuído por várias aplicações, plataformas, linguagens, tecnologias e fontes de dados diferentes.

Considerando para fins de estudo neste artigo, uma aplicação corporativa *"X"*, simplesmente denominada de *CIA (Central de Integração Avançada)*, que diariamente necessita *capturar*, *processar* e *disponibilizar* informações para os seus usuários. De forma resumida, essa tarefa parece ser simples, mas expandindo a visão um pouco mais, pode-se perceber um nível mais granular de etapas a serem cumpridas, sendo: (1) capturar dados de um Webservice; (2) converter os dados para um formato estruturado; (3) transformar os dados em informações úteis para o contexto; (4) aplicar lógica de negócios; (5) validar as regras de negócios; (6) gerar novas informações; (7) armazenar as informações processadas; (8) enviar as informações por e-mail; (9) disponibilizar as informações para os sistemas de inteligência de negócios (Business Intelligence - BI), dentre outros destinatários; e (10) notificar a equipe que o fluxo concluiu com sucesso. Se expandirmos a visão para cada etapa citada, iremos perceber um nível maior de pequenas atividades dentro de cada etapa principal.

Analisando o fluxo acima percebe-se que é necessário executar várias etapas ou atividades para colocar a informação a disposição do usuário. Partindo do pressuposto que esta solução de software pode estar baseada em uma arquitetura de *Microserviços*, onde cada etapa a ser cumprida pode estar em uma aplicação diferente, em locais distribuídos, utilizando tecnologias e/ou linguagens de programação diferentes, pois cada etapa pode possuir uma natureza distinta que exige soluções diferentes para se obter um melhor resultado, é fácil perceber o alto nível de complexidade requerido para executar todo o fluxo de trabalho de forma eficaz e eficiente. Ser eficaz nesse tema se refere ao fato de disponibilizar a informação correta, enquanto ser eficiente se refere ao fato de disponibilizar a informação no tempo exigido pelo usuário.

Controlar esses fluxos de trabalhos e garantir que todas as rotinas necessárias para cumprir determinada tarefa ou etapa sejam executadas com sucesso é uma tarefa complexa e desafiadora, que exige um esforço de arquitetura e programação que pode extrapolar os limites de conhecimento, tempo e custos de uma empresa.

Nesse contexto, a *Teoria dos Grafos* pode fornecer um modelo de conjuntos denominado de *DAG (Directed Acyclic Graph ou Grafo Acíclico Direcionado)*, formado por seus elementos e suas relações, fornecendo uma alternativa para criação, organização e visualização de fluxos de trabalho. Esse modelo proporciona uma visão clara das etapas a serem cumpridas e permitem aos desenvolvedores a criação de estratégias para lhe dar com a complexidade da orquestração destes elementos.

Dessa forma, este artigo tem como objetivo apresentar a solução de software *Apache Airflow*, como alternativa para gerenciar fluxos de trabalho complexos, através da implementação de grafos do tipo *DAG*.

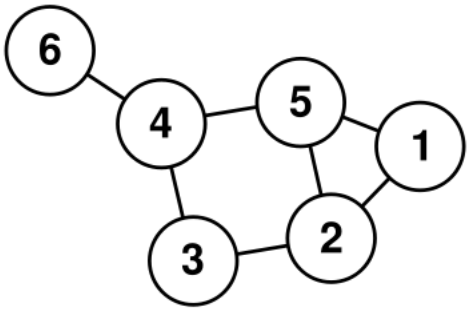

Fig. 1. Exemplo de Grafo

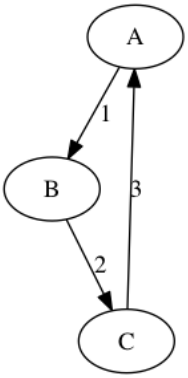

Fig. 2. Grafo Cíclico Direcionado

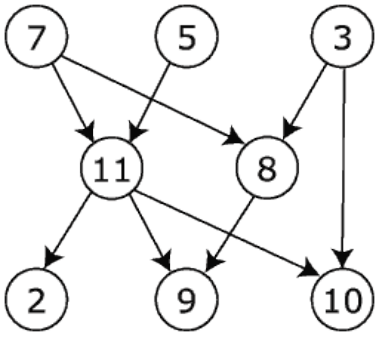

Fig. 3. Grafo Acíclico Direcionado

2 Grafos

Um *Grafo* pode ser entendido como um conjunto de elementos ou nós, denominados *Vértices "V"* que são unidos por *Arestas "A"* (ver figura 1). Logo um *grafo* pode ser representado pela notação *G(V, A)*. A *Teoria dos Grafos* é um ramo da matemática que estuda as relações entre estes objetos.

Dependendo do caso, as arestas podem ou não ter uma direção definida. Uma aresta pode ligar um vértice a ele próprio ou não. Caso uma aresta tenha uma direção, representado graficamente por uma seta, este indica que grafo é do tipo direcionado ou orientando, podendo ser chamado ainda de Dígrafo. As arestas podem ainda ter um peso numérico associado a elas.

2.0.1 Grafos acíclicos direcionado

Um *grafo acíclico direcionado* ou simplesmente *DAG*, é um grafo dirigido sem ciclo, ou seja, para qualquer um dos vértices do grafo, jamais haverá uma ligação dirigida que inicie e termine no mesmo vértice, por isso a denominação, acíclica ou sem ciclo. Esse tipo de grafo é utilizado em um contexto onde não faz sentindo que um elemento tenha uma ligação com si próprio ou onde não possa haver ciclos.

Um bom exemplo para este caso é a árvore de dependências ou bibliotecas dos sistemas operacionais com base no *Unix*, como é o caso do *Ubuntu* e *Debian*. Ao instalar uma determinada biblioteca, o sistema automaticamente instala todas as dependências necessárias para o seu funcionamento. Observando a figura 3 percebe-se que a biblioteca *"A"* depende da biblioteca *"B"*, e esta por sua vez, depende da biblioteca *"C"* que depende de *"A"*. Se for solicitado ao sistema operacional a instalação da biblioteca *"A"*, possivelmente ocorrerá um *loop* infinito, pois ao instalar a biblioteca *"C"*, a biblioteca *"A"* seria instalada novamente reiniciando o fluxo de trabalho.

Um modelo de elementos baseado em DAG garante que nunca exista um caminho que saia de um determinado vértice e chegue nele mesmo, conforme mostra a figura 2. O modelo fornecido por um DAG encaixa perfeitamente no cenário de fluxos de trabalhos de sistemas de informação que necessitam executar etapas com uma direção única dentro de uma mesma instância do processo, garantindo que o fluxo tenha um inicio, meio e fim, mesmo que haja caminhos alternativos a serem seguidos, além do fluxo principal.

3 Apache Airflow

Em tempos que a demanda por informação é muito grande, várias são as fontes de dados e diversas arquiteturas que os desenvolvedores necessitam trabalhar. Muitas vezes os dados estão distribuídos em banco de dados, arquivos (csv, txt, logs), API's de terceiros, dentre outros. A resposta para esse problema é a consolidação dos dados e dos fluxos de trabalho, e para que isso possa ocorrer é de fundamental importância um *"orquestrador"*, pois em muitos casos algumas informações devem ser processadas primeiro que outras, seguindo uma sequencia lógica, para só assim no final, obter-se uma consolidação correta de informações para atingindo o que se espera.

Nesse cenário de coordenar *fluxos de trabalhos* surge o *Airflow* que é uma ferramenta de código aberto para controlar fluxos de trabalho computacionais complexos, com poder de interagir com os dados entre várias etapas do processo. O projeto teve inicio em outubro de 2014, sendo criado por *Maxime Beachemin* no *Airbnb* e servia para controle de fluxo de trabalho interno. Em junho de 2015 teve seu código aberto sob o *Github da Airbnb* e desde março de 2016 é mantido pela fundação *Apache*.

A plataforma *Airflow* é uma ferramenta que descreve, executa e monitora fluxos de trabalho, por meio da criação de um *Grafo Acíclico Direcionado (DAG)* que mantém as tarefas organizadas de tal maneira que não existam quebras de seus relacionamentos e dependências. Um fluxo de trabalho em *Airflow* é representado como um DAG (ver figura 4) que pode conter diversas tarefas independentes. Pode ainda possuir restrições de caráter temporal ou que necessite

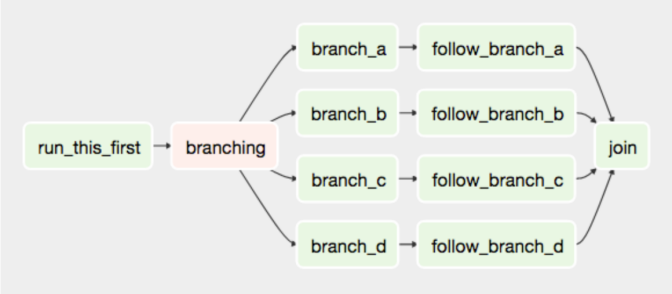

Fig. 4. Exemplo de DAG no Airflow

garantir que alguma tarefa anterior tenha sido executada corretamente.

Nesse sentido, temos um exemplo bem claro na própria documentação do *Airflow* quando cita que:

> "Um DAG simples pode consistir em três tarefas: A, B e C. Pode-se dizer que A precisa ser executado com êxito antes que B possa ser executado, mas C pode ser executado a qualquer momento. Pode-se dizer que a tarefa A expira após 5 minutos e B pode ser reiniciada até 5 vezes no caso de falha. Pode também dizer que o fluxo de trabalho será executado todas as noites às 22h, mas não deve começar até uma determinada data."

A forma do grafo representa a funcionalidade geral do fluxo de trabalho, que pode conter várias ramificações, identificadas como tarefas ou etapas. O modelo pode definir se todas as tarefas serão executadas ou se alguma será ignorada, podendo inclusive definir qual tarefa ignorar em tempo de execução.

3.0.1 Instalando e Configurando o Airflow

O *Airflow* é desenvolvido em *Python*, por tanto, os arquivos de configuração e definição das DAG's também são escritos em *Python*. A instalação e configuração do *Airflow* pode ser realizado seguindo alguns poucos passos. O *Airflow* pode ser executado com *Python* versão 2.7 ou versão 3.6, sendo esta última a mais recomendada. Para instalar as seguintes etapas devem ser cumpridas, sendo:

- Configurar variável de ambiente para a pasta *home* do *Airflow*, que irá armazenar o banco de dados, arquivos de configurações e fluxos (DAG's):

```
$ export AIRFLOW_HOME=~/airflow
```

- Caso necessário, pode-se criar um novo ambiente virtual para o *Python* e suas dependências:

```
$ pip install virtualenv
$ virtualenv airflow
$ cd airflow/bin
$ source activate
```

- Instalar o *Airflow*:

```
$ pip install airflow
```

- Inicializar o banco de dados do *Airflow*:

```
$ python airflow initdb
```

- Iniciar o servidor do *Airflow*:

```
$ python airflow webserver -p 8080
```

- Para verificar se o *Airflow* está executando corretamente, basta acessar o endereço via navegador:

```
http://localhost:8080
```

Para utilização em ambiente de testes o *Airflow* cria uma base de dados local do tipo *SQLite*. Para utilização do *Airflow* em ambientes de produção recomenda-se a utilização de outros bancos de dados como *PostgreSQL* ou *MySQL*. O banco de dados padrão vem populado com alguns exemplos de DAG's que podem ser úteis para aprendizado sobre o funcionamento da plataforma.

3.0.2 Criando DAG's

A criação de novas DAG's segue um processo simplificado. Basta escrever um script *Python* e armazená-lo na pasta *airflow_home/dags*. Abaixo é apresentado um exemplo simplificado de um script *Python* que implementa uma DAG para o fluxo de trabalho citado no início do artigo. O exemplo completo pode ser visualizado diretamente no repositório do *GitHub*[1]:

```
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator \
import DummyOperator
from airflow.operators import BashOperator

dag = DAG(
    'carga_diaria_faturamento',
    description='Carga Diária de Faturamento',
    schedule_interval='0 12 * * *',
    start_date=datetime(2017, 3, 20),
    catchup=False
)

capturar_webservice = DummyOperator(
    task_id='tarefa_capturar_webservice',
    retries=3,
    dag=dag
)

...

notificar_conclusao = BashOperator(
    task_id='tarefa_notificar_conclusao',
    bash_command='echo "Notificando..."',
    dag=dag
)

capturar_webservice >> notificar_conclusao
```

O código acima cria um fluxo de trabalho do *Airflow*, segindo a seguinte lógica: (1) importa-se as bibliotecas básicas do *Airflow*; (2) cria-se o objeto *"dag"* que irá representar e conter a definição do fluxo; (3) cria-se o operador (operação/tarefa/etapa) representando pelos objetos

1. Exemplo completo do fluxo de trabalho do Apache Airflow https://github.com/bal699/airflow-sample

"capturar_webservice" e *"notificar_conclusao"*; (4) por último, indica-se qual o fluxo (direção) de execução das atividades, através do operador do tipo *bitshift* do *Python* representado pelo síbolo *">>"*. Em seguida, será apresentado mais detalhes sobre elementos disponíveis no *Airflow*.

3.0.3 Operadores

O DAG de forma geral determina como o fluxo de trabalho será executado. Enquanto isso um *"operador" (operação)* determina o que de fato será executado. O operador pode ser visto como etapas/tarefas/atividade isoladas que executam de fato a lógica do fluxo de trabalho, como: capturar um arquivo, converter dados de um arquivo *.csv* para um tabela no banco de dados, etc.

Os operadores devem ter uma característica chamada de *"idempotência"*, ou seja, devem ter a capacidade de serem executados de forma repetitiva, sem gerar resultados diferentes da execução inicial. Os operadores também devem ser capazes de executar de forma autônoma, ou seja, um operador deve ser capaz de executar sua função de forma independente. Dessa forma, qualquer etapa do fluxo pode ser executado a qualquer momento, obedecendo logicamente suas restrições de precedência, permitindo uma recuperação exatamente onde o fluxo foi interrompido em casos de falhas ou executar fluxos alternativos com base em decisões em tempo de execução, sem afetar os resultados esperados.

O *Airflow* possui um conjunto de operadores comuns, como: *BashOperator* - para executar comandos bash; *PythonOperator* - para realizar chamadas a funções *Python*; *EmailOperator* - para enviar e-mails; *HTTPOperator* - para enviar uma solicitação HTTP; *MySqlOperator*, *SqliteOperator*, *PostgresOperator*, *MsSqlOperator*, *OracleOperator*, *JdbcOperator* - para executar comandos SQL; dentre outros. Ainda é possível criar operador personalizados de acordo com a necessidade.

Cada operador possui um conjunto de parâmetros que definem como será o seu comportamento, como: *dag* - indica a DAG que o operador pertence; *retries* - indica quantidade de vezes operador deverá ser executado antes de falhar; *schedule_interval* - segue a mesma sintaxe de programação de um agendamento *cron* e indica qual o intervalo de execução do operador; *start_date* - indica quando a tarefa iniciará a execução, dentre outras configurações.

Um operador pode ser atribuído a um DAG no momento da sua criação ou posteriormente. Porém, é importante destacar que uma vez que um operador é atribuído a um DAG, este não poderá ser transferido ou desatribuído.

3.0.4 Relacionamento

Os relacionamentos do *Airflow* cumprem o papel das *Arestas* do modelo DAG. Estes relacionamentos, ligações e sequencias entre os operadores, são definidos pelo método *"set_downstream()"* ou pelo operador *">>"*, e pelo método *"set_upstream()"* ou pelo operador *"<<"*. A cadeia de fluxos é sempre executada da esquerda para direita. O código abaixo define o relacionamento entre os operadores e são equivalentes:

```
op1 >> op2
op1.set_downstream(op2)
```

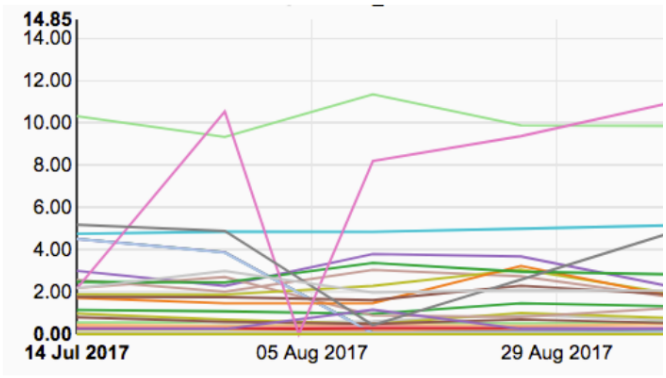

Fig. 5. Gráfico de Tempo do Airflow

```
op2 << op1
op2.set_upstream(op1)

op1 >> op2 >> op3 << op4
op1.set_downstream(op2)
op2.set_downstream(op3)
op3.set_upstream(op4)
```

Os relacionamentos podem ser realizado de uma para mais operadores ao mesmo tempo, bastando utilizar o formato de listas do *Python*, como no exemplo a seguir:

```
branching >> [branch_1, \
branch_b, branch_c, branc_d] >> join
```

O código acima gera uma DAG similar ao apresentado na figura 4, onde o operador *branching* possui uma ligação quadrupla para os operadores *branch_**.

3.0.5 Instâncias de Tarefas

Os códigos apresentados anteriormente representam de forma abstrata o modelo do fluxo de trabalho e tarefas que serão invocadas. Porém, o que será executado na prática, são instâncias de cada tarefa, que são a combinação de um DAG, uma tarefa e um ponto no tempo. Cada instância possui parâmetros e valores específicos em relação do seu modelo abstrato e representa uma execução da tarefa. Cada instância, portanto, irá representar um *"nó"* em uma DAG. Cada instância conterá um *status* que indica a sua situação dentro do fluxo de trabalho, como: *"em execução"*, *"sucesso"*, *"falhou"*, *"ignorado"*, *"para tentar novamente"*, dentro outros.

3.0.6 Gerenciamento do Fluxo de Trabalho

Para facilitar o gerenciamento dos fluxos de trabalhos e instâncias configuradas o *Airflow* oferece alguns recursos como *"gráfico de tempo de duração" (figura 5)* e *"gráfico de gannt" (figura 6)* que mostra o tempo de execução de cada etapa do processo e permite identificar qual a operação mais demorada no fluxo de trabalho, facilitando a identificação de pontos críticos e tarefas que podem ser paralelizadas.

Este recurso torna-se interessante para ambientes em que a lógica das operações evoluem e tornam-se mais complexas

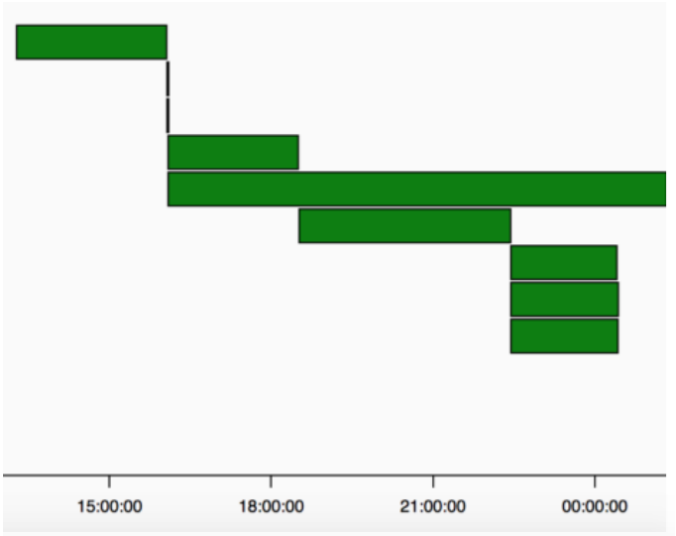

Fig. 6. Grafico de Gannt do Airflow

com o passar do tempo, permitindo o rastreamento de e identificação de gargalos, observando e comparando as execuções das tarefas ao longo do tempo.

O correto alinhamento dos tempos de execução é de extrema importância para garantir que a informação seja processada de maneira eficiente e disponibilizada para o usuário no tempo necessário, pois para a grande maioria dos negócios, o atraso da informação pode causa um grande impacto no processo de tomada de decisão.

4 Conclusão

Como pode-se observar, o *Airflow* oferece uma infraestrutura pronta para criação e gerenciamento de fluxos de trabalho complexos através da implementação de modelos baseados em grafos do tipo DAG. A combinação de DAG's, operadores e instâncias, permite a criação e gerenciamento de fluxos de trabalho complexos, facilitando a vida dos desenvolvedores que não precisam empregar esforço para criar tal infraestrutura.

O *Airflow* fornece: (1) ponto central de programação de fluxos; (2) flexibilidade na integração de diferentes tarefas; (3) possibilidade de distribuir e escalar a si próprio; (4) comunidade de código aberto que permite colaboração e evolução rápida da ferramenta.

A solução pode ser utilizada em ambientes de softwares corporativos, com dados centralizados ou distribuídos, arquiteturas baseadas em microserviços, ou sistemas monolíticos que necessitam garantir que os fluxos de trabalho sejam executados de maneira eficaz (de forma correta) e eficiente (no tempo necessário).

A utilização da linguagem *Python* pode fornecer um ponto positivo para o desenvolvimento, pois simplifica e remove algumas barreiras relacionado a sintaxe durante a escrita do código, proporcionando mais rapidez na configuração dos fluxos.

Por fim, com *Airflow* é possível remover a complexidade de fluxos baseados em diversos scripts e componentes produzidos em diferentes linguagens, centralizando e fornecendo uma visão coesa de dependências, onde pode-se visualizar de forma clara cada etapa a ser executada, fornecendo gerenciamento e controle.

References

[1] A. Rocha, "Seus dados em um único lugar com Airflow - Desenvolvimento de software sob medida," 2017. [Online]. Available: http://dextra.com.br/pt/blog/seus-dados-em-um-unico-lugar-com-airflow/

[2] L. Bhatt, "Airflow - Beginners Tutorial," 2016. [Online]. Available: https://tech.lalitbhatt.net/2016/04/airflow-beginners-tutorial.html

[3] Apache Software Foundation, "Airflow Documentation." [Online]. Available: https://airflow.apache.org/license.html

[4] M. Karzyński, "Get started developing workflows with Apache Airflow - Michał Karzyński," 2017. [Online]. Available: http://michal.karzynski.pl/blog/2017/03/19/developing-workflows-with-apache-airflow/

[5] "Apache Airflow." [Online]. Available: https://www.etltools.net/apache-airflow.html

[6] D. Tran, "How Sift Trains Thousands of Models using Apache Airflow - Sift Science Engineering Blog : Sift Science Engineering Blog," 2018. [Online]. Available: https://engineering.siftscience.com/sift-trains-thousands-models-using-apache-airflow/

[7] K. Pedersen, "Creating Dynamic Tasks Apache Airflow — Official Pythian Blog," 2018. [Online]. Available: https://blog.pythian.com/creating-dynamic-tasks-using-apache-airflow/

[8] M. Beauchemin, "Airflow: a workflow management platform – Airbnb Engineering & Data Science – Medium," 2015. [Online]. Available: https://medium.com/airbnb-engineering/airflow-a-workflow-management-platform-46318b977fd8

[9] G. Amigo, "Tornando dados acessíveis em uma arquitetura distribuída e de microsserviços," *Conferência Internacional de Desenvolvimento de Software*, 2017. [Online]. Available: https://www.linkedin.com/in/gustavoamigo https://twitter.com/gustavoamigo https://github.com/gustavoamigo

Eduardo Luiz Bacharel em Administraçao pela Universidade Estadual do Rio Grande do Norte (UERN). Mestrando em Engenharia de Software pelo Instituto Metrópelo Digital (IMD - UFRN). Trabalha como coordenador do setor de desenvolvimento de softwares da empresa Avance - Inteligência em Gestão. Desenvolve sistemas de integração de dados baseado em fluxos de trabalho.

Marcelo Luiz Possui graduação em Direito pela Universidade Estadual da Paraíba (2008), é Tecnólogo em Analise e Desenvolvimento de Sistemas pela Fundação Universidade do Tocantins e é Especialista em Segurança da Informação pela Faculdade de Tecnologia de João Pessoa. Atualmente é técnico em informática da Universidade Estadual da Paraíba, com atribuições de gerenciamento da rede local e manutenção dos computadores do Centro de Humanidades em Guarabira-PB e Gerente de Tecnologia no Provedor de Internet voax.com.br, com atribuições na manutenção dos serviços prestados, prevenção, tratamento e respostas a incidentes. Tem experiência na área de Ciência da Computação, com ênfase em Redes de Computadores e Segurança da Informação.